Synthesisable Recursion for C++ HLS Tools

David B. Thomas
Dept. of Electrical Engineering, Imperial College London
dt10@imperial.ac.uk

***Abstract*—C-based HLS tools continue to improve in analysis and optimisation, but are still restricted to a subset of C functionality. A C language feature missing from all common HLS tools is recursion, which makes it difficult and time consuming to write many types of programs in HLS. This paper presents a technique for implementing recursion as an Embedded Domain Specific Language (EDSL) in C++, utilising the C++ front-end of a HLS compiler to build the state machines and stacks, while ensuring the code presented to the back-end is completely synthesisable. While the EDSL is not pure C, it provides a user-friendly language that can be trivially translated from a recursive C program, and can contain calls to plain C for compute-intensive leaf functions. HLSRecurse is a platform independent library that allows the same program to be compiled using g++, clang, Legup, or Vivado HLS, and have the same execution semantics. The performance of HLSRecurse is evaluated in software and hardware using three micro-benchmarks of the underlying state-machine builder, and five practical examples of real recursive programs including a Sudoku solver, Strassen multiplication, and adaptive Monte-Carlo integration. We show that for Vivado HLS the DSL provides the same area-time product as manually converted programs, while in Legup the DSL increases the area-time product by 1.5x.**

I. Introduction

While C-based HLS tools have existed for decades, a combination of robust commercial and open-source academic compilers, together with the need to address the design gap for larger FPGAs, have led to a resurgence in interest and use. The modern tools provide greatly improved analysis and optimisation of code, but support for C language constructs has remained at about the same level – you are allowed to use anything from C, except recursion, function pointers, and dynamic memory.

This paper presents a method for adding recursion to existing HLS tools, without requiring compiler modifications or third-party tools. We take advantage of the fact that the front-end language of modern HLS tools is C++ rather than C, and create a compiler independent library for writing recursive functions. The library is pure C++, but is designed to use only features which will be synthesisable by a HLS back-end. Recursive functions are written in an embedded Domain Specific Language (DSL) for describing control-flow, which closely resembles the original recursive C implementation, and can contain fragments of plain C or calls to other functions.

The main contributions of this paper are:

- A technique for building composable state machines using C++ which can be synthesised by HLS tools.
- A platform independent architecture and compilation approach for supporting recursive synthesised programs.
- An embedded DSL for describing recursive programs, allowing a single recursive program to be compiled to both software and hardware.
- An evaluation of the DSL over eight recursive programs using Legup and Vivado HLS, showing that the DSL has a resource and execution time within a factor of two of manually converted recursive programs.

The library works in off-the-shelf Legup and Vivado HLS, and includes both the test programs and all scripts needed to replicate the results presented here.

II. Background

Recursion occurs when a function makes a call to itself, either directly or indirectly via another function. Recursion can be used to implement control-flow constructs such as iteration and selection, and from a theoretical view the recursive Lambda calculus is equivalent in power to Turing machines [5]. Many functional and high-level languages, including Haskell and Scheme, choose recursion over iteration for purity and expressiveness reasons, but lower-level languages such as C and C++ prefer to provide both.

Iteration is easier for a compiler to analyse than recursion and generally better understood by programmers [4], so most high-performance code emphasises iteration. Recursion is usually only found when it is needed for algorithmic reasons, such as using a divide-and-conquer algorithm to reduce running time from $O(n^2)$ to $O(n \log n)$. Usually a balance is struck which uses iteration in a computationally intensive base-case, while less efficient recursion is used to reduce algorithmic complexity at a high-level.

C-based HLS tools have traditionally not supported recursion, and it is explicitly disallowed in tools such as Legup [1], Vivado HLS [8], and Handel-C [2]. The lack of support for recursion appears to be due to a lack of demand from end-users, plus the lack of a program stack. Nothing precludes the addition of a stack to these tools, but it would require significant changes to the underlying compilation strategies.

Strategies for supporting recursion have been proposed, tackling both RTL and early HLS languages [7] These focussed on the architectures to be built and how to organise them, leaving the implementation as a manual process to be applied. Further work extended this to recursive processing of dynamic data structures [6], but in the process is still very manual – in both cases the number of applications considered was small, due to the time and complexity of applying the strategies to real programs.

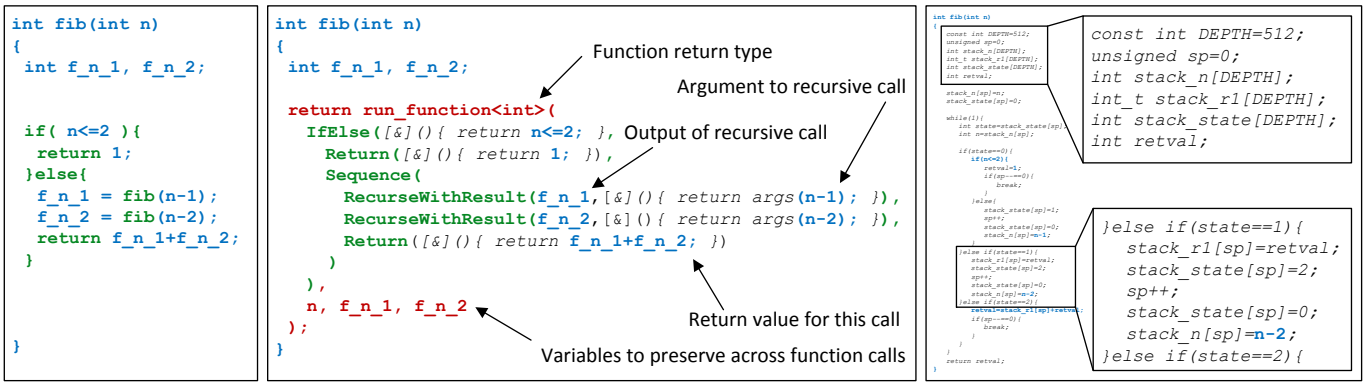


Fig. 1: Fibonacci code using true recursion, synthesisable EDSL recursion, and synthesisable manual recursion.

Another approach to recursion has come from the functional synthesis community, who attempt to map pure and impure functional languages to hardware. Verity is an impure functional language, and supports synthesisable recursion via combinators [3]. Another approach uses rewriting techniques to lower Haskell to SystemVerilog [9], and so has good support for recursion. However, both approaches are more concerned with expressibility and pushing the limits of HLS, rather than robustness and the creation of production-quality hardware.

What is missing is an approach which allows for the high-level recursion needed for algorithmic complexity, while still reaping the benefits of a production-quality optimising synthesis tool to provide high-performance leaf functions. Current approaches provide either robust HLS tools and loop synthesis but no recursion, or experimental HLS tools with poor quality of results. As a result, the typical HLS user encountering recursion must manually remove any recursion before synthesis, requiring longer to test and bug, and resulting in less maintainable code.

III. Design Goals

The idea of this work is not to try to find limited or special-case solutions, but to develop techniques that can work in many languages, platforms, and applications. The main four goals can be summarised as follows:

- Readable: functions should be immediately understandable, and resemble recursive C code.
- Editable: it should be possible to locally modify a recursive function without requiring global changes to function control flow, or to move well-formed fragments of code between recursive functions.
- Portable: applications using the library should work in both software and hardware compilers with no source changes or conditional compilation.
- General: it should be possible to express the types of recursion found in a wide array of programs.

These goals have been met, and are illustrated in Figure 1, which shows three versions of a fibonacci function: on the left is a native recursive (non-synthesisable) C function; in the middle is a synthesisable function using the C++ DSL presented here; and on the right is a synthesisable function which was manually converted to a stack plus state-machine C implementation.

While the DSL version does not look exactly like the original, it is clearly understandable, and it is also possible to edit it by changing the function locally. As we will go on to show, the code is also portable and can be compiled in both software and two current HLS tools. The generality of the approach will be demonstrated by showing it can be applied to eight different recursive applications exhibiting different types of recursion. To achieve this, three main challenges need to be solved: building composable state machines at C++ compilation time; creating a synthesisable architecture for recursion in HLS-oriented C++; and creating a DSL which allows users to describe recursive programs on top of the state machines and architecture.

IV. Declarative State Machines

A difficulty in embedding a DSL for use in HLS is the restriction on code constructs that can be used. Specifically, we must ensure:

- Pointers and references used within the DSL are statically resolvable at compile-time, to ensure that the HLS tool can make intelligent register allocations, and can use a "points-to" analysis to identify distinct local RAMs.
- No pointer casts, virtual methods, or function pointers.
- Syntactic sugar used to describe programs can be completely optimised away by the C++ compiler and language independent optimisation passes, so that the HLS back-end only sees C-level constructs.
- No run-time recursion (as that is what we are trying to add!), though bounded compile-time recursion is allowed.
- It is possible to embed "native" code within the DSL, as performance sensitive leaf code must not incur any overhead from the DSL.

C++11 provides a number of constructs which can help, and after experimentation, it was determined that both Vivado HLS and Legup will accept the following constructs:

- **Compile-time Meta-Programming**: Any computation which can be expressed as a template meta-program can be used to *guarantee* that there is no run-time overhead. Legup naturally supports this, as it works only on LLVM IR, but VHLS performs front-end (C-level) annotations – fortunately, all pure template meta-programs passed through, possibly due to an internal reliance on template meta-programs to implement some math functions.
- **Functors**: Invocable objects with state (functors) can be passed to templatised functions, and can be completely optimised away at the call site. Often this will allow member variables of the functor to be effectively lifted to local variables in the enclosing scope, then passed by reference to an inlined version of the functor body.
- **By-Reference Lambdas**: C++11 lambdas are a language feature which allows anonymous functions to be locally declared and stored in a variable. An important feature of lambdas is that it allows the function to "close over" any variables in the surrounding environment: if a by-reference lambda refers to a variable `x` in the current environment, then when `x` is modified inside the lambda it will also be changed outside the lambda. Lambdas are essentially syntatic sugar for functors, but are much more convenient for the programmer.

Support for lambda and functor inlining is particularly important in this work, in order to make a program statically analysable. Figure 2a shows the de-sugaring of a construct from left to right. The first panel shows a lambda function capturing `x` by reference, and then storing the lambda in the variable `f`. Whenever and wherever `f(.)` is invoked, the code within the lambda will modify the original variable `x`. Internally the C++ compiler will de-sugar the lambda to a functor, shown in the next stage, which contains a reference to `x` and a function which contains the lambda body. Because the type of the functor is statically known, during optimisation the class can be stripped away, leaving just the reference and the body. Further compiler optimisation passes can statically replace the reference with the target variable and inline the body, leaving the HLS-supported form shown in the last panel.

Armed with this knowledge, we can now create a DSL for state machines, which allows static analysis of the state transition graph at compile-time, while also allowing blocks of code to be embedded within the state machines. Rather than an arbitrary state machine, the DSL only allows structured programming constructs, mirroring the control constructs found in C: sequencing, alternation, and iteration.

The basic building block of the DSL is a *step*, shown at the top of Figure 2b. A *step* is any functor with an integer property `state_count`, and a function `step`. The property `state_count` must be a compile-time integer, and defines the total number of distinct sub-states in the step. A step could contain user code in the form of a lambda function, in which case it contains just one state with code zero. However, the step might also be a composite step, such as a sequence step, in which case the `state_count` is the sum of the count of the contained states.

Given an instance of a step, it is possible to invoke the functionality of any of the contained sub-states by calling `step` with the index of the current state. The step object will then execute any logic associated with the current state, and return the next state that should be executed. Using a linear integer state identifier offers a number of advantages:

- A caller can initiate any step by passing in state 0, and can determine the step has finished when `step` returns a value outside [0,`state_count`).
- Composite steps can adjust state identifiers before passing them on to the sub-steps, laying out the state range of the sub-steps linearly within the range of the composite step.
- Template meta-programs can examine and manipulate identifiers at compile-time, performing optimisation of the state machine.
- State identifiers are small positive integers, which are very fast to compare (reducing state machine critical path), and can be cheaply stored in order to remember where a particular computation was paused.

The lower part of Figure 2b shows a *SequenceStep*, one of the primitives provided by the DSL. This step can contain two arbitrary sub-steps, and its only purpose is to make sure that the first step is executed before the second step. The first step may jump through many (or an infinite) number of states before it finishes, but when it does, the sequence step will map the following state to the start of the second step. Looking at the sequence step definition it is clear that it can be statically desugared into an if statement. If the two sub-components can also be statically expanded, then an arbitrary graph of steps can be recursively de-sugared into a nested set of `if` conditions by the C++ compiler.

The DSL provides a number of primitive steps for constructing control flow graphs, as well as a set of helper functions for constructing the graphs. The grammar of the language is very simple, but allows any structured program to be described:

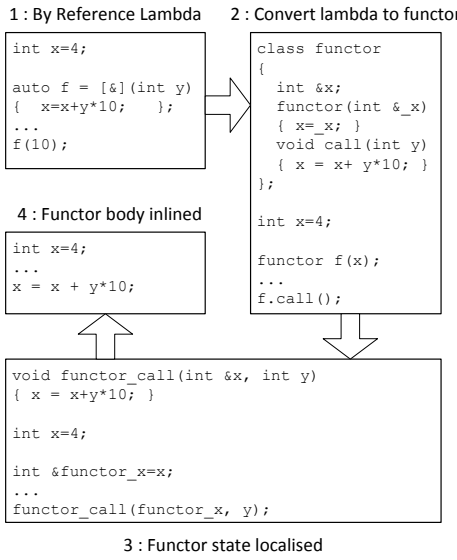
```
STAT ::= Sequence ( STAT_LIST )
       | If ( COND , STAT_LIST )
       | IfElse ( COND , STAT, STAT )
       | While ( COND , STAT )
       | Pass ( )
       | Break ( )
       | Continue ( )
       | Functor ( ACTION )
       | ACTION

STAT_LIST ::= STAT
            | STAT , STAT_LIST

TYPE    ::= C++ type (e.g. void, int, bool)
ACTION  ::= lambda of type void()
COND    ::= lambda of type bool()
```

This state machine provides the control structures shown in green in Figure 1, while the user code shows as blue actions within lambda functions.

The steps provided in the state machine DSL library have a slightly more complex interface than shown here, which enables more state-machine operations to be performed at compile-time. In particular it supports the passing of arbitrary compile-time traits to the step function, allowing sub-steps



(a) De-sugaring of an anonymous lambda function (1) to synthesisable C-level statements (4).

```
template<class TFirst, class TSecond>
struct SequenceStep
{
  TFirstStep m_firstStep;
  TSecondStep m_secondStep;

  // TFirst and TSecond are any kind of step object
  SequenceStep(
    const TFirst &fStep,
    const TSecond &sStep)
  : m_firstStep(fStep), m_secondStep(sStep)
  {}

  // Compile-time constant
  enum{ state_count // We contain...
      = TFirst::state_count // all of TFirst,
      + TSecond::state_count // then all of TSecond
  };

  // Step by a single state, returning the next
  // state to execute, or state_count to finish
  unsigned step(unsigned state) const
  {
    if(state < TFirst::state_count){
      // In the range mapped to first step
      return m_firstStep.step(state);
    }else{
      // In the range mapped to second step
      // Convert to state space of second step
      unsigned local=state - TFirst::state_count;
      local=m_secondStep.step(local); // Invoke it
      // Convert back to outer state space
      return local + TFirst::state_count;
    }
  }
};
```

(b) The generic interface for a step (top), and an implementation of a sequencing step.

```
int s1;
char s2;
double s3;

const int D=64;
int sp;
u32_t stk[4*D];


void push(){
  stk[sp++]=(u32_t)s1;
  stk[sp++]=(u32_t)s2;
  stk[sp++]=lo(s3);
  stk[sp++]=hi(s3);
}

void pop(){
  u32_t hi=stk[--sp];
  u32_t lo=stk[--sp];
  s3=mk_double(hi,lo);
  s2=(char)stk[--sp];
  s1=(int)stk[--sp];
}
```

```
int s1;
char s2;
double s3;

const int D=64;
int sp;
int    stk_s1[D];
char   stk_s2[D];
double stk_s3[D];

void push()
{
  stk_s1[sp]=s1;
  stk_s2[sp]=s2;
  stk_s3[sp]=s3;
  sp++;
}

void pop()
{
  --sp;
  s1=stk_s1[sp];
  s2=stk_s2[sp];
  s3=stk_s3[sp];
}
```

(c) Sequential deep word-oriented stack (left), versus parallel shallow context-oriented stack (right).

to inherit attributes from their containers. This is used to support detection and exploitation of tail-recursion, and is also used to provide diagnostics for the user, such as causing a compile-time error when a `Break()` is not enclosed within a `While(.)` step.

The state machine DSL is useful by itself, as it provides a clean and portable to use way of representing control structures which can be statically expanded to HLS compatible control structures. However, its primary role is to support recursion, as discussed in the next section.

V. Recursion as a Library

In order to implement recursion, it must be possible for more than one invocation of a function to exist at the same time. However, only one invocation will be active at any one time, and the state of other functions will be held on the stack. The state machine DSL provides a way to manage the currently active function, as it provides a means of scheduling lambda functions providing by the user. The active function can interact directly with function variables – for example, the Fibonacci function in Figure 1 contains user lambdas which read and write `f_n_1` directly. So when a recursive function call is made, the current value of the variable must be automatically saved to the stack, then restored when the recursive call finishes.

To implement recursion we will first determine an efficient stack architecture to support saving and loading of the active variables in an FPGA, and then extend the DSL with features needed to support recursion primitives. Finally the augmented state machine and stack architecture are brought together to provide full recursion support.

A. Stack Architecture

Software stacks are implemented using what is logically a narrow but very deep RAM, typically one word wide, and potentially millions of words deep. Recursive functions must save their stack by sequentially writing the registers in the current frame to this RAM, and will sequentially load the stack back into registers when the child call returns. This arrangement offers a lot of flexibility, as for example a function may write different amounts of data to the stack depending on the active state across each recursive call, but presents a potential performance bottleneck. Many architectures provide hardware optimisations to mitigate this sequential register-memory bottleneck, such as rotating register files or specialised stack instructions, while the L1 cache tends to keep the most recent contexts hot.

FPGAs provide many on-chip RAMs which are typically around 32-bits wide and 512-1024 elements deep. Physical RAMs can be joined to make larger logical RAMs, either by placing physical RAMs in parallel to make a logical RAM with a wider data-bus, or by using chip-select logic to make a deeper RAM with the same width data-bus. The HLS tool and/or synthesis tools will perform the creation of the logical RAMs, on our behalf, but it is still necessary to choose the stack management model.

Using a software-style word-oriented stack would allow the ability to optimise stack space needed across each recursive function call, but has three drawbacks. First, it is extremely difficult to perform deep analysis without compiler help, so for the EDSL approach there is no variable liveness information available. Second, a single stack would require many registers to be routed to the stack input and output ports, causing extra wiring and potential congestion. Third, and more importantly, sequential access to a shared RAM will mean that pushing

and popping the function context will take time linear in the function context size. This bottleneck is show in the left side of Figure 2c, where each push or pop will take 2 cycles using a dual port RAM, or 4 cycles with a single port RAM.

For this work, the stacks are made as wide and shallow as possible, maximising available stack bandwidth. For each variable in the function context a stack is created with the same type as the variable, as shown on the right-hand side of Figure 2c. This code arrangement makes it clear to the toolchain that the stack transactions are independent, and also that each variable is associated with exactly one stack. This allows the HLS compiler to trivially schedule all stack operations in parallel, no matter how large the function context is, and means that at the synthesis stage it is clear that the RAM inputs and outputs are associated with exactly one register. The unique variable-stack mapping does not mean the output of the stack RAM is wired directly into the register input, as there must be other writers to the variable (otherwise why is it being saved?), but significantly reduces the fanout and routing compared to a shared stack.

B. Recursion Primitives

There are four main primitives needed in order to support recursive calls:

- *Start* a new function call using given parameter values.
- *Suspend* execution of the currently active function, in order to start a recursive call.
- *Resume* execution of a previously suspended function, optionally retrieving the result of the recursive call.
- *Complete* a function call, optionally returning a result.

By definition any executed function must *Start*, but there is no guarantee about whether any or all of the other primitives will be executed – a function may not make a recursive call, in which case it is never *Suspended* or *Resumed*, while some functions may deliberately enter an infinite loop and never *Finish*.

Because the DSL approach to recursion cannot perform any static analysis, we must assume the worst: the active function might spontaneously perform a recursive call at any point, requiring a *suspend* to capture the state and push it on the stack. As a consequence, we must also be prepared for the function to eventually *resume* execution at any point. Fortunately, the DSL approach from the previous section makes this problem more manageable, as it reduces the number of places where a recursive call might happen.

If we consider a function body expressed using the state machine DSL, then it is possible that a user may perform a recursive call within a lambda. But the entire point of this work is that HLS tools cannot currently implement recursion, and so will flag the function call as invalid. As a consequence, we can guarantee that recursive function calls will not arise within the state-machine DSL (though non-recursive function calls may).

Recursive calls can now be integrated into the state machine in a manageable way, by making recursion a special kind of `Step`. When a `RecurseStep` is executed, it will:

1) Push the value of the function variables onto the corresponding stacks, capturing the active state.
2) Push the state code for the state following the `Recursestep` onto the stack, recording where execution will resume.
3) Evaluate a user supplied lambda function `RecurseStep::argSrc` which is stored within the step object, creating a tuple of function arguments.
4) Assign the tuple of function arguments to the function variables, overwriting the suspended function's context.
5) Return a special state code, telling the recursion manager to go to the first state of the function body.

Once the Recurse step is complete, the old function call has been suspended and the new function call has been made the active function.

While these steps are shown as sequential, in hardware there is great potential for parallelism. For example, we have already decided to have many parallel stacks, one for each variable. In fact, if the user supplied lambda executes in a single cycle then *every* stage of the call can be performed in parallel, and this fact should be statically visible to the HLS tool. So in principle, the cost of a function call is as low as a single cycle.

Function returns are handled as another type of special step, following a reverse process. The main difference is that there is a user lambda for supplying the return value, rather than function arguments. As with the Recurse step, the parts of the Return can all execute in parallel, so the limiting factor on execution time is the user supplied return value function.

C. Recursion DSL

In order to expose the recursion architecture to end-users, the recursion architectures and steps need to be embedded in the state-machine DSL. The C++ type names of the recursion step objects quickly become very complicated, so a set of constructor functions is used to avoid needing to mention types.

```
STAT' ::= STAT
        | Recurse ( ARG_SRC )
        | Return ()
        | RecurseWithResult( RET_VAR , ARG_SRC )
    | Return ( RES_SRC )

ARG_SRC ::= lambda returning tuple(...)
RET_VAR ::= reference to a variable
RES_SRC ::= lambda returning scalar result value
```

The two different forms of Recurse and Return correspond to functions that do or do not return values, as limitations in C++ mean that a user must explicitly specify which variable should receive the result of a non-void function.

A recursive function created using this DSL is simply a data-structure, which describes the control-path as well as all the variables used during execution. To actually execute the function a wrapper called `run_function` is used, which is the part highlighted in red in Figure 1. This function is responsible for creating the stack and initialising the state machine. It then runs the state machine `step` function in a

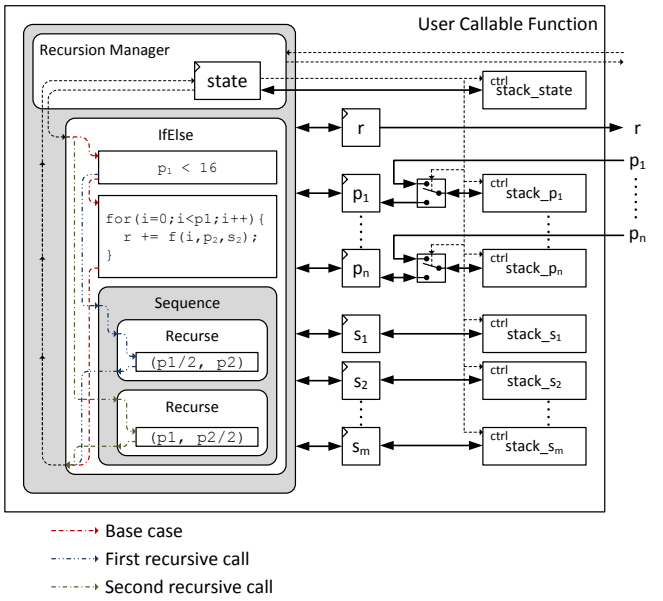


Fig. 3: Hardware structure of a recursive function, with steps of the state-machine on the left, and stacks on the right.

loop, responding to the special codes for *Suspend*, *Resume*, and *Return*. If a *Return* code happens when the stack is empty then the recursive function is finished, and `run_function` returns to the original caller.

Due to limitations in C++ compile-time introspection, the user must specify the variables which must be persisted across function calls. These include the function parameters, for example `n` in Figure 1, and any local variables which must be preserved across function calls, such `f_n_1` and `f_n_2`. The total number and types of persisted variables is unlimited, and the DSL will automatically infer how many stacks are needed and what types they should be.

The overall architecture built by `run_function` is shown in Figure 3, showing a state machine on the right, and the stacks on the right. On startup the parameters $p_1..p_n$ will be retrieved from the current environment and pushed onto the stack. The recursion manager will then send the initial state into the state machine, following one of the dashed paths. Certain paths will cause a recursion step to execute, at which point the entire context tuple $(p_1, .., p_n, s_1, .., s_n)$ will be pushed onto the stacks.

The two code-blocks at the top of the IfElse block are the condition and true-body of the conditional. Both are specified as lambdas, and so will be compiled as plain C by the HLS compiler, while still having read-write access to the current context. Any code within these lambdas will run at full-speed and can use all standard optimisation `#pragmas`.

VI. Evaluation

As recursion is not supported in current main-stream HLS tools, there is currently no baseline for measuring performance. In order to address this we first present a benchmark suite of eight recursive C programs. The benchmark is then used to examine the efficiency of the proposed approach, by comparing manually de-recursed implementations to recursive implementations using the DSL. The comparisons are made in terms of resource requirements and execution time, looking at post place-and-route performance in Virtex-7 using Vivado HLS and Cyclone-V using Legup.

A. Benchmark Test-Cases

Two main types of test-cases are considered: three micro-benchmarks, which consist almost entirely of recursive calls; and five applications, which are real-world recursive applications. The applications are chosen to do something "useful", in the sense that they actually solve problems. In all cases the original algorithm came from existing software-oriented C code found in the wild, and was not written or re-written specifically to target HLS *except* as necessary to support recursion. It is not claimed that these are optimal hardware solutions - in some cases, such as FFT and sorting, the approach used is clearly far from optimal, whether in HLS or RTL. The assumption here is that a designer has been given existing recursive code or an algorithm to port to HLS, so we are interested in the relative performance of manual rewriting versus using the DSL.

Fibonacci: A naive dual-recursive fibonacci:

$$\text{fib}(n) = \begin{cases} 1, & \text{if } n <= 2 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{otherwise} \end{cases} \quad (1)$$

The amount of compute per function call is minimal, allowing the approximate cost per function call to be determines.

The Ackerman function: This is a function from computability theory which demonstrates huge growth in the number of recursive invocations for small input arguments. This is an example of a non primitive-recursive function, as it is not possible to identify a parameter that strictly decreases with each function call. Like the Fibonacci function it contains little compute and is bottlenecked by function call overhead

Heap sum: Given a vector $x_1..x_n$ of integers, calculate the prefix-sum of the nodes in a binary heap. This splits down to individual elements and updates the array in place, so it provides an example of a recursion and memory intensive program with little arithmetic intensity.

QuickSort: This uses quick-sort to sort an array in-place, using recursive quicksort while there are 32 or more elements still to be sorted, then switching to an iterative merge-sort for less than 32 elements.

FFT: A radix-2 fixed-point Fast Fourier Transform, using 32-bit fixed-point complex numbers. The implementation is deliberately recursion heavy, immediately splitting down to 2 point transforms, but also contains an iterative merge.

Strassen: The Strassen matrix multiplication decomposes the multiplication of two $n \times n$ matrices into the cross multiplication of eight $n/2 \times n/2$ sub-matrices. By performing an additional $O(n^2)$ additions and subtractions on the sub-matrices, it only requires 7 smaller $O(n^3)$ recursive multiplications. These sub-multiplications can themselves be

implemented using Strassen multiplication, resulting in an asymptotic complexity of $O(n^{\log_2 7})$.

MISER: MISER is an algorithm for adaptive Monte-Carlo integration, which is able to explore a multi-dimensional integrand and concentrate samples in the areas of highest variance. It uses recursion in order to partition the volume of integration: at each level it randomly samples the current volume, splits the volume along the axis with the highest variance, and allocates the remaining sample budget to the two sub-volumes according to their variances. Unlike the previous examples this algorithm has a very unpredictable recursion pattern, as it may end up with a reasonably balanced binary tree of volumes, or the recursion tree may degenerate and become unbalanced as MISER explores a difficult region.

Sudoku: Sudoku can be solved using many methods, but one of the simplest is a recursive backtracking solver, which simply chooses an unknown cell, and iterates through each possible assignment. When a possible binding is found, the solver makes a tentative binding, then recurses to the next cell. When no binding can be found for the current cell, the solver backtracks to the previous binding.

B. Performance Results

The eight benchmarks were translated to two synthesisable forms: "manual", where the author manually translated all recursive calls to stack pushes and pops; and "DSL", using the DSL presented in the paper. The method used to get to the manual version was the same semi-mechanical (though not automated) process each time. The manual conversion was designed to minimise the human effort involved, while still ending up with a similar underlying parallel stack architecture for all eight programs.

The two programs were then compiled and synthesised using two different tool-chains: Vivado HLS and Vivado 2015.2, targetting Virtex-7; and Legup-4.0 and Quartus 14, targeting Cyclone-5. In both cases the synthesised programs were checked against the original software unit-tests for the programs, ensuring the RTL code is still correct. The designs were then placed-and-routed to get resource utilisation, while cycle-accurate simulations were used to find the execution time for different problem sizes. From this point we use the short-hand VHLS to refer to the Vivado/Xilinx flow, and Legup to refer to the Legup/Altera flow.

Resource utilisation is shown in Table I, broken down into the key resource types. The right-most group gives the resource usage of the DSL relative to the manual version, with values greater than 1.0 identifying where the DSL requires more resources. The upper group targets VHLS, while the lower group targets Legup.

Overall the DSL abstraction appears to incur a relatively small abstraction penalty for most programs, but there is a large amount of variability. Sometimes the DSL is faster, and sometimes manual, though overall there seems to be some penalty for the DSL. The biggest outlier is the FFT when executing in VHLS, where the DSL requires more than 6x the number of LUTs, but in exchange uses fewer DSPs and block-RAMS. Examining the output of the tools gives no specific insight into *why* the differences occur - both tools see the DSL and manual versions as completely different programs (which they are), and compile them to very different state machines.

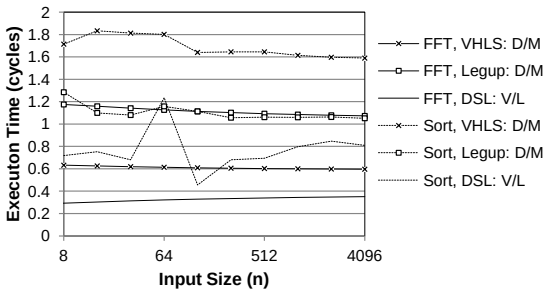


Fig. 4: Relative execution time for Sort and FFT applications as n increases. D/M = DSL/Manual. V/L = VHLS/Legup.

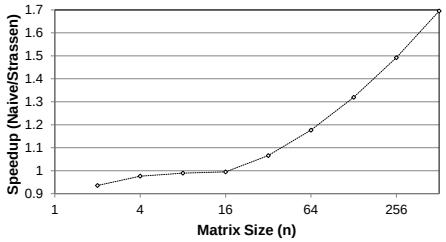


Fig. 5: Speedup of Strassen over naive multiplication.

As well as resource usage, we also need to compare the run-time of the different algorithms. All the benchmarks except Sudoku take a run-time parameter n specifying the size of the problem, and the total cycles required scales with n according to the intrinsic complexity class of the algorithms. Figure 4 compares the execution time of the two $O(n \log n)$ applications as n is increased. The square box lines show the performance for Legup, and both applications are slightly faster in the manual version, though this decreases with n. In comparison, the crossed lines show that with VHLS there are big differences, with the DSL running Sort around 1.6x faster than manual, but the FFT is slowed down to 0.6x.

An argument for using recursive functions is that often they allow for more sophisticated divide-and-conquer algorithms with better asymptotic run-times. Figure 5 shows the measured speed-up for matrix multiplication, comparing the recursive Strassen against a naive $O(n^3)$ matrix multiplication. As expected, for small matrices, the naive method is slightly faster, as it incurs none of the recursive overhead. But even for n=32 the Strassen multiplier is able to save one 16x16 base multiplication, providing a small speed-up, and this speed-up rapidly grows.

Table II summarises the absolute latencies of each program in cycles for a fixed per-problem n.[1]. The D/M column then

[1]Miser is excluded for Legup due incorrect simulation results, apparently due to a problem in the Legup generated RTL. The program simulates but gives the wrong answer, so the cycle count cannot be fully trusted.

		DSL				Manual				DSL/Manual			
	Design	LUTs	FFs	DSPs	RAMs	LUTs	FFs	DSPs	RAMs	LUTs	FFs	DSPs	RAMs
VHLS	Fibonacci	348	307	0	3	309	272	0	3	1.13	1.13	-	1.00
	Ackerman	287	335	0	1.5	298	298	0	2	0.96	1.12	-	0.75
	HeapSum	294	186	0	1.5	306	191	0	1.5	0.96	0.97	-	1.00
	QuickSort	858	759	0	2	631	592	0	2	1.36	1.28	-	1.00
	FFT	32767	3145	35	4	5250	5229	61	4.5	6.24	0.60	0.57	0.89
	Sudoku	1476	1486	0	2	1167	1121	0	2	1.26	1.33	-	1.00
	Strassen	6398	5601	3	8.5	5851	5219	3	7.5	1.09	1.07	1.00	1.13
	MISER	7311	5824	22	6	7820	6847	70	4	0.93	0.85	0.31	1.50
	GeoMean	1657.60	1104.42	13.22	2.94	1222.36	1084.94	23.40	2.89	1.36	1.02	0.56	1.02
Legup	Fibonacci	687	998	0	14	409	535	0	6	1.68	1.87	-	2.33
	Ackerman	287	335	0	1.5	942	1764	0	6	0.30	0.19	-	0.25
	HeapSum	629	1091	0	12	545	609	0	10	1.15	1.79	-	1.20
	QuickSort	2053	3525	0	18	1897	3120	0	12	1.08	1.13	-	1.50
	FFT	3935	6123	3	47	3834	5515	37	31	1.03	1.11	0.08	1.52
	Sudoku	2261	3642	0	18	1660	2693	0	10	1.36	1.35	-	1.80
	Strassen	9838	12201	2	80	12097	10574	2	16	0.81	1.15	1.00	5.00
	MISER	17880	22861	81	74	16013	23634	27	58	1.12	0.97	3.00	1.28
	GeoMean	2113.73	3075.17	7.86	19.70	2169.67	3005.67	12.60	13.70	0.97	1.02	0.62	1.44

TABLE I: Post place and route resource utilisation for benchmark programs.

		Time (cycles)			Area	T× A
		DSL	Manual	D/M	D/M	D/M
VHLS	Fibonacci	31005	31005	1.00	1.08	1.08
	Ackerman	46207	41233	1.12	0.93	1.05
	HeapSum	45053	49150	0.92	0.98	0.90
	QuickSort	243172	152993	1.59	1.20	1.91
	FFT	507881	851943	0.60	1.18	0.70
	Sudoku	107183	87860	1.22	1.19	1.45
	Strassen	19112268	18929389	1.01	1.07	1.08
	MISER	5132855	5054677	1.02	0.78	0.79
	Geomean	309326.4	301813.0	1.02	1.04	1.07
Legup	Fibonacci	152411	90415	1.69	1.94	3.27
	Ackerman	246279	175331	1.40	0.24	0.34
	HeapSum	233426	163808	1.42	1.35	1.93
	QuickSort	300284	285501	1.05	1.22	1.29
	FFT	1447811	1349554	1.07	0.61	0.66
	Sudoku	209758	158321	1.32	1.49	1.98
	Strassen	9154658	9145875	1.00	7.13	7.14
	MISER	-	15360188	-	1.43	-
	Geomean	495319.4	621475.6	1.26	1.28	1.61

TABLE II: Comparison of execution latency in cycles. D/M = DSL/Manual. Area is geometric mean of all resource types.

gives the ratio of the DSL to manual time, showing that the execution times of the two are very similar. In order to assess the area-time tradeff, the table includes a crude assessment of relative area, using the ratio of the geometric mean of all four resource types (LUT,FF,DSP,BRAM). We can see that relative area and time are not well correlated, as sometimes the DSL is larger and slower, and sometimes it is larger but faster. The final column shows estimated area-time product, showing that on average the DSL and manual versions are about the same under VHLS, but for Legup there does appear to be on average an abstraction penalty of around 50%.

VII. Conclusion

This paper presented an embedded Domain Specific Languages for supporting recursion in existing HLS tools with a C++ front-end. Recursion is supported by defining a synthesisable state machine representation which can be built and manipulated by the C++ compiler. This is combined with a parallel stack architecture for implementing recursion at runtime, and syntactic support for easily describing recursive programs. The resulting library allows recursive programs to be compiled and executed using current HLS tools, with no platform-specific extensions or suport code. The library is demonstrated across eight benchmark programs, showing that the DSL is competitive with manually converted recursive code: sometimes better, sometimes worse, but within a factor of two in both resource usage and execution time.

The current DSL is suitable for use with production compilers, but there remain a number of possible improvements. One is to improve the end-user experience, by providing better error reporting, and supporting a richer set of constructs. Another is to improve performance, by performing more optimisations of the state machines and stacks at compile-time.

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, Sept. 2013.

[2] Celoxica Ltd. *Handel-C Language Reference*, 1999.

[3] D. R. Ghica, A. Smith, and S. Singh. Geometry of synthesis iv: Compiling affine recursion into static hardware. *SIGPLAN Not.*, 46(9):221–233, Sept. 2011.

[4] C. M. Kessler and J. R. Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, 2(2):135–166, 1986.

[5] S. C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.

[6] S. Ninos and A. Dollas. Modeling recursion data structures for fpga-based implementation. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 11–16, Sept 2008.

[7] V. Sklyarov. FPGA-based implementation and comparison of recursive and iterative algorithms. *Microprocessors and Microsystems*, 28(5):197–211, 2004.

[8] Xilinx. Vivado design suite user guide: High-level synthesis (ug902 (v2014.2). http://www.xilinx.com, 2014.

[9] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–93. IEEE Press, 2015.